

Ted M. Young

Java Trainer, Coach, Live Coder, and Creator of JitterTed's TDD Game



The Event Sourcing Mindshift

Simple Patterns = More Flexible Systems

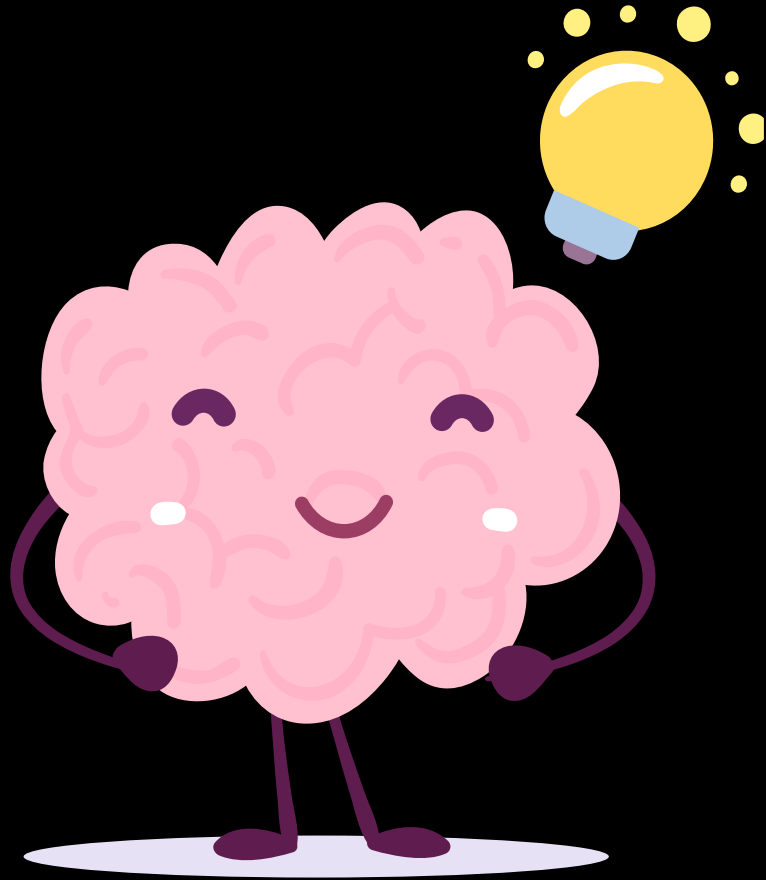
Me: <https://ted.dev/about>

Source Code? Slides?

<https://ted.dev/talks>



Ask Questions As You Need



I may defer the
answer if I'm going
to cover it later!



What's in this TED Talk...

1. JitterTicket: a concert ticketing domain
2. Persisting decisions: ORMs & audit tables
3. Fundamentals: **Commands**, **Events**, **Projections**
4. Where do decisions live? Deciders & aggregates
5. Benefits, trade-offs & when **not** to use it
6. Next steps



JitterTicket Concert Ticketing

The Event-Sourced Ticketing System

Explore Concert Tickets

Event Viewer

Concert Sales



JitterTicket Domain

CUSTOMER

CONCERT

TICKET



What is APPLICATION STATE?

Recording Outcome of DOMAIN DECISIONS

TIME

Domain Decisions

- **Purchase Ticket**

- **Decision (Domain Rule): enough tickets?**
 - Yes -> State Change: Ticket Purchased
 - No -> No State Change (inform buyer)

- **Reschedule Concert**

- **Rules: not canceled; in the future; no schedule conflicts**
 - Yes -> State Change: Concert Rescheduled
 - No -> No State Change (report why)



How Do We Persist Decision Outcomes?

Often: Map Objects to Database Tables (ORM)

Mapping Objects from/to Database



r/ExperiencedDevs
u/Fuzzy_World427 · 10h

DDD: How do you map DTOs when entities have private setters?

Hey all,

I'm running into trouble mapping DTOs into aggregates. My entities all have **private setters** (to protect invariants), but this makes mapping tricky.

I've seen different approaches:

- Passing the whole DTO into the aggregate root constructor (but then the domain knows about DTOs).
- Using mapper/extension classes (cleaner, but can't touch private setters).
- Factory methods (same issue).
- Even AutoMapper struggles with private setters without ugly hacks.

So how do you usually handle mapping DTOs to aggregates when private setters are involved?



24

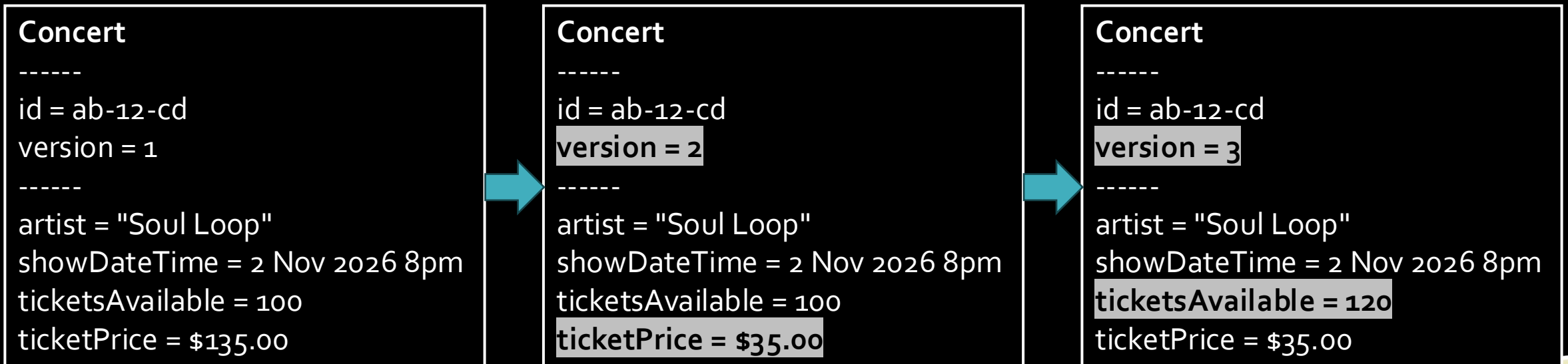


51



Storing Data as "Latest State"

- "State-Sourced"
- No history, throws away old information
- No knowledge of how/why state changed



"But we have an Audit Table"

Guess About Future Needs

- ☐ Need questions in advance
- ☐ Guess wrong? Too late: data's gone.



Audit Table: Lossy by Design

What We Knew

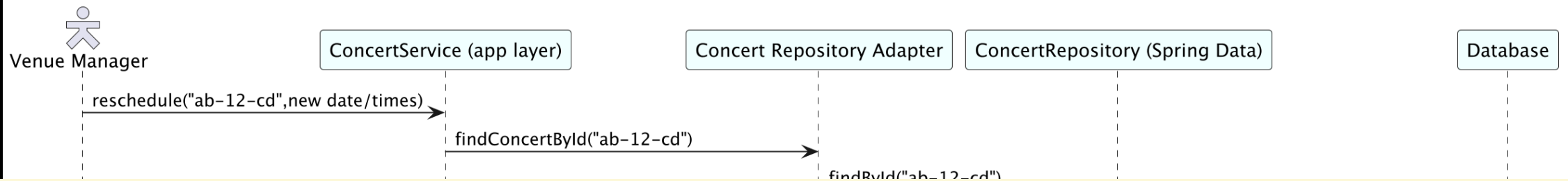
- actor (initiator)
- full request
- clock
- state as of decision
- which rule triggered
- the outcome

What We Stored

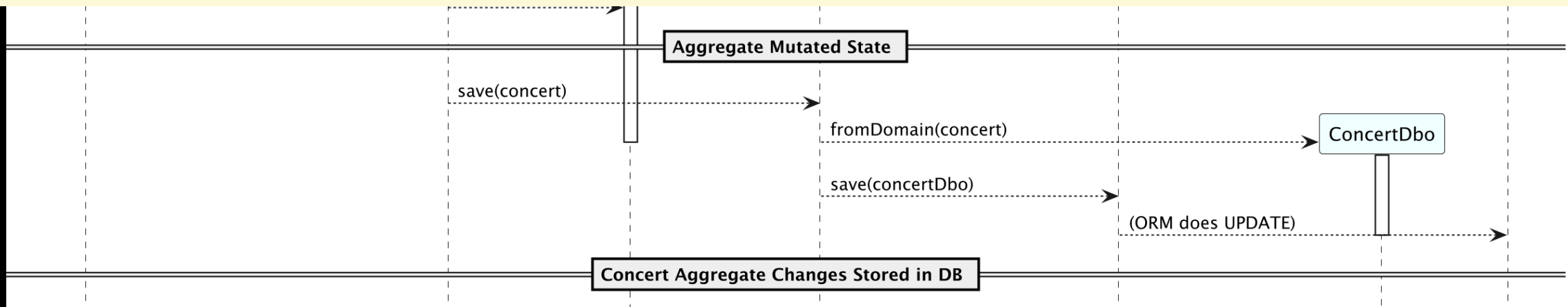
- previous value
- new value
- timestamp
- user



Adapter-Driven ORM Persistence



**FAMILIAR
BUT COMPLICATED**



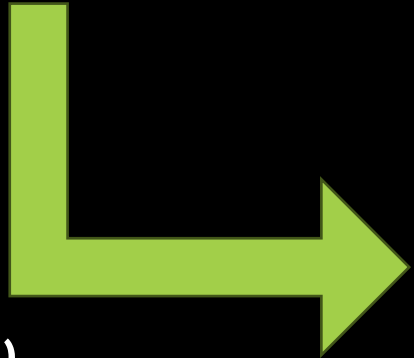
Storing Data as Changes in State

Concert Scheduled: 23, ab-12-cd, "Soul Loop", 2 Nov 2026, 20:00,
100, 135.00

Ticket Price Changed: 41, ab-12-cd, 35.00

Capacity Increased: 73, ab-12-cd, 120

*Apply
("fold")
Events*



Concert

id = ab-12-cd

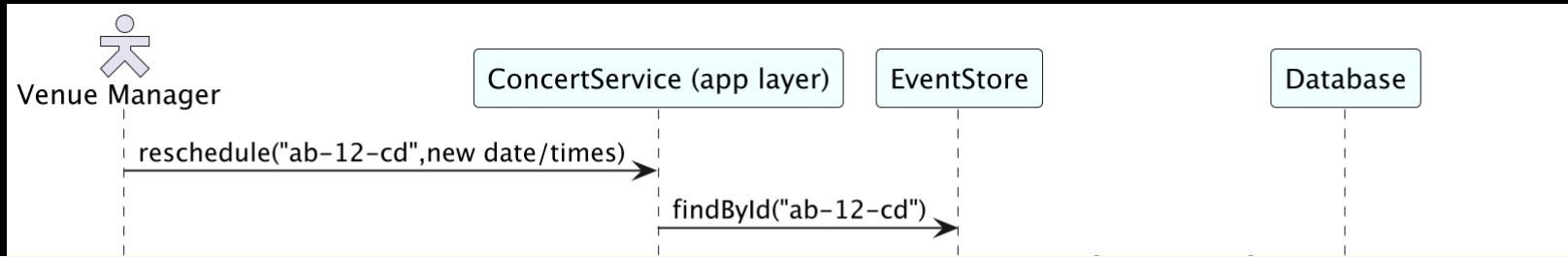
artist = "Soul Loop"

showDateTime = 2 Nov 2026 8pm

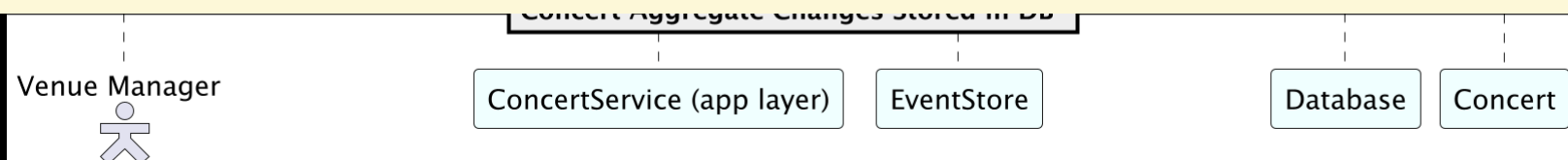
ticketsAvailable = 120

ticketPrice = \$35.00

Event-Sourced Persistence (Aggregate-based)



**SIMPLER
BUT LESS
KNOWN**



Fundamentals

Past

Event

Present

State

Future

Command



Definition: EVENT (the past)

a fact, something that happened

Customer Registered

Concert Scheduled

Ticket Price Changed

Capacity Increased

Tickets Sold



Definition: STATE (now)

Relevant Application info used to *DECIDE*

Concert

id = ab-12-cd

version = 1

artist = "Soul Loop"

showDateTime = 2 Nov 2026 8pm

ticketsAvailable = 100

ticketPrice = \$135.00

In the future?

No conflict
with other
Concerts?

Enough tickets
available?

How much to charge?



Definition: COMMAND (future)

Request to change state

Register New Customer

Schedule Concert

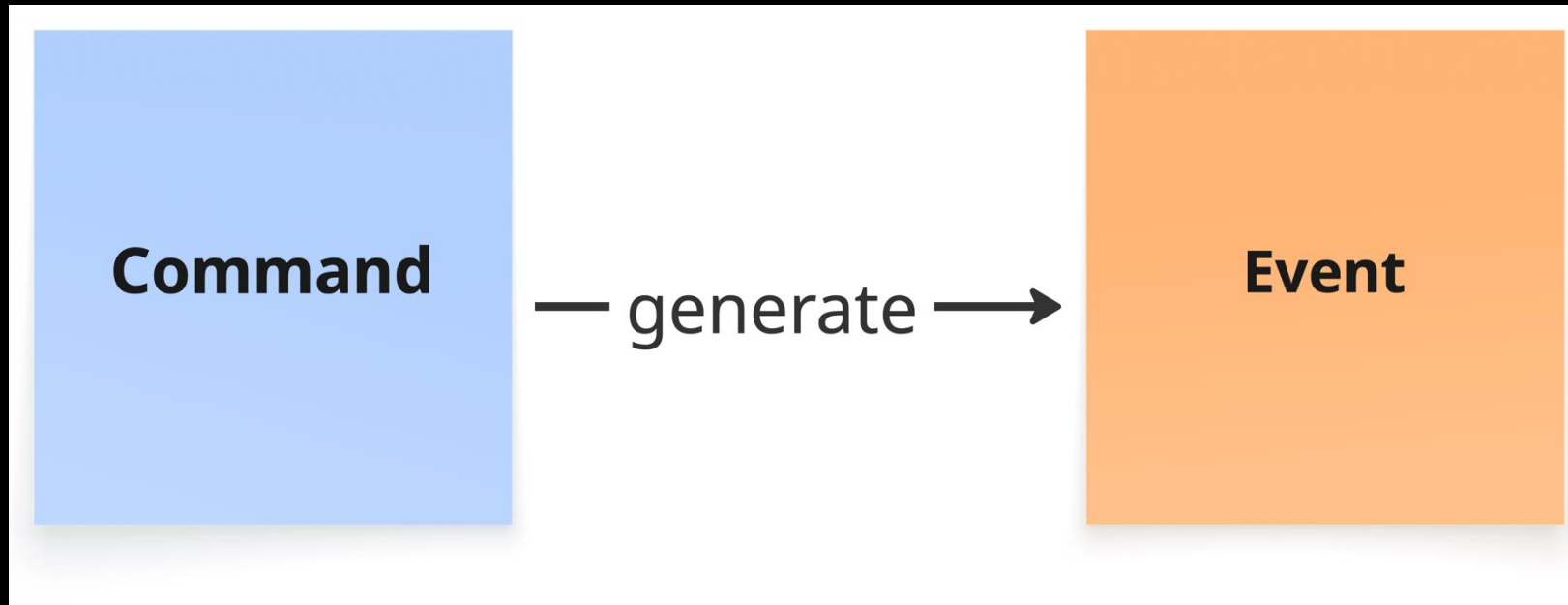
Change Ticket Price

Increase Capacity

Purchase Tickets



Commands Generate Events



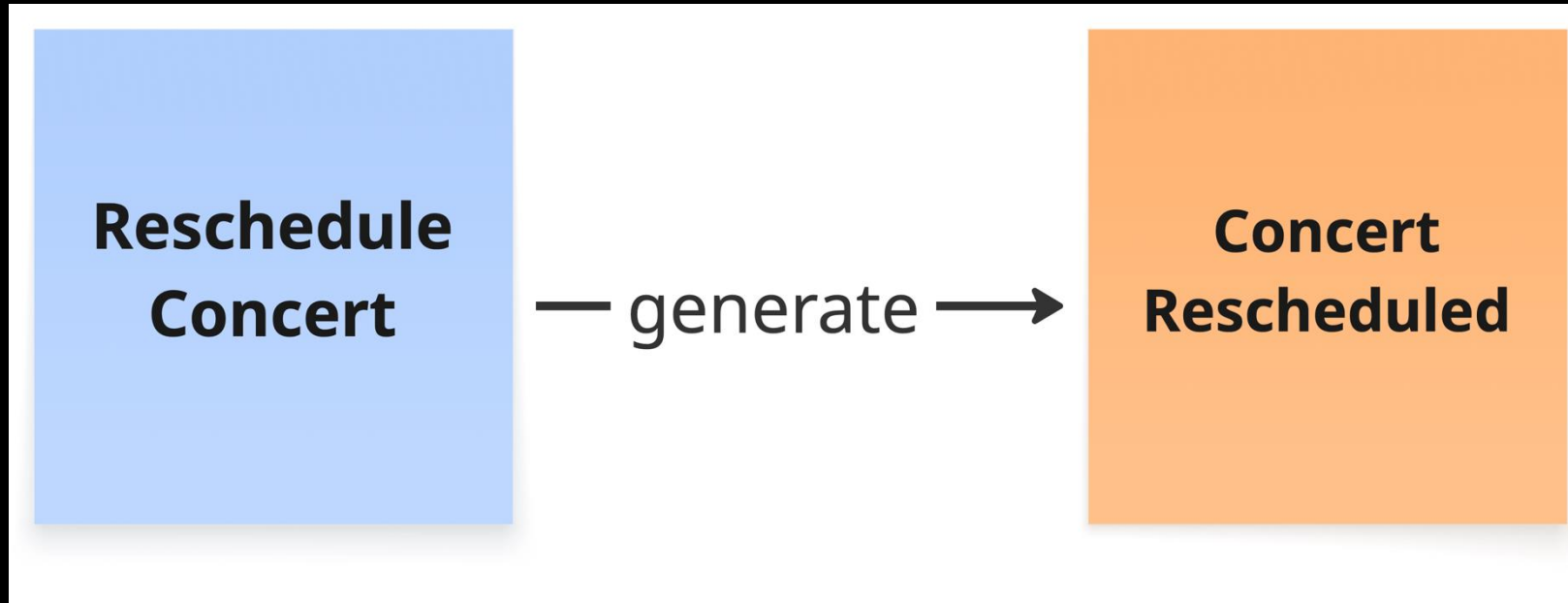
Command

Event

State



Commands Generate Events



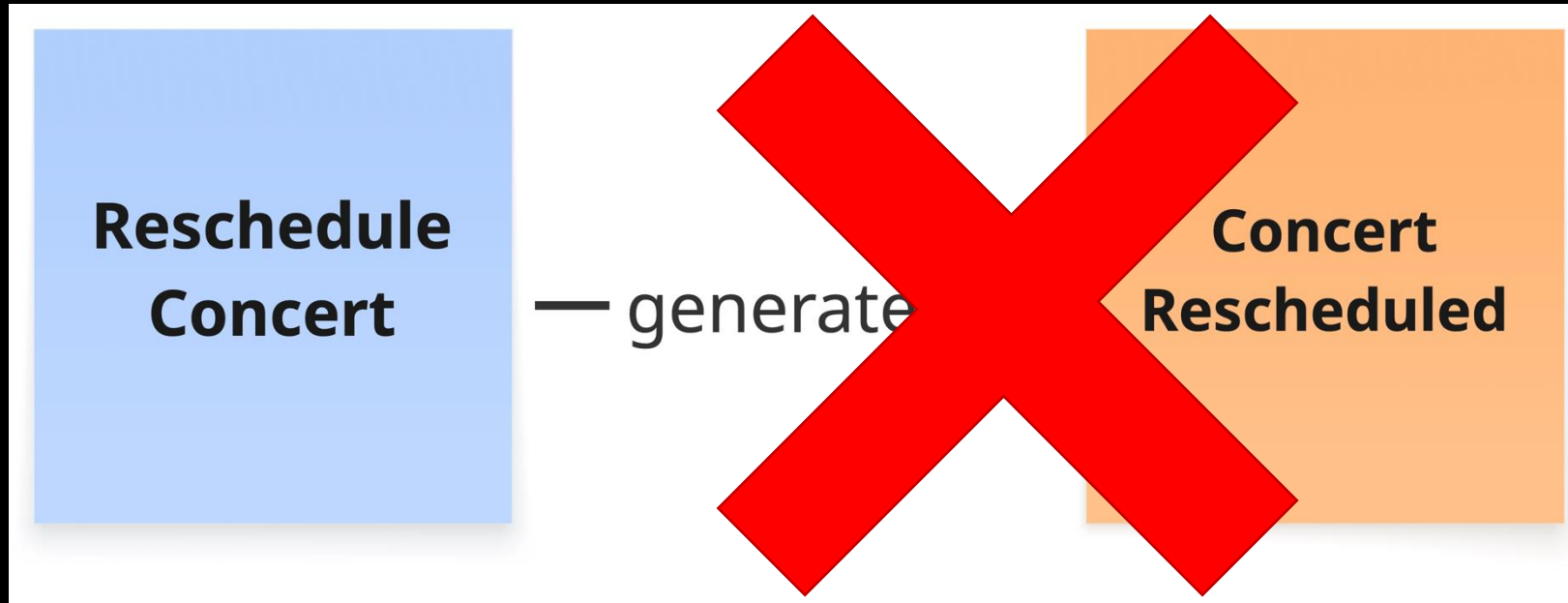
Command

Event

State



...unless It's Not Allowed (Rule)



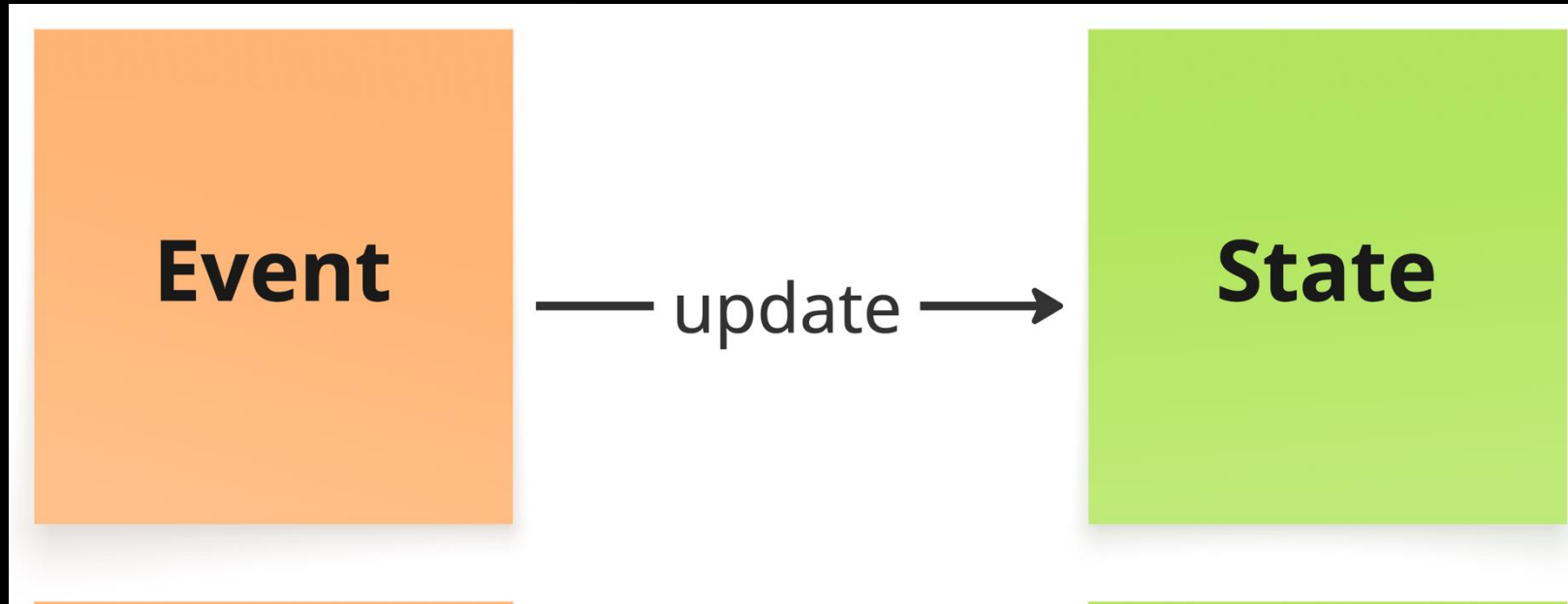
Command

Event

State



Events Update State



Command

Event

State



Events Update State (always!)



Command

Event

State



Events *Project* to State



Command

Event

State



Definition: PROJECTOR

Function that constructs a *Read Model* (state view) by sequentially *replaying events* (fold) loaded from the *Event Store*.

Produces a READ MODEL



```

public class AvailableConcerts implements ReadModel {
    public void apply(SequencedEvent sequencedEvent) {
        lastSequenceNumber = sequencedEvent.sequenceNumber();
        switch (sequencedEvent.event()) {
            case ConcertScheduled scheduled -> {
                concerts.put(scheduled.concertId(),
                    new Concert(scheduled.concertId(),
                        scheduled.artist(),
                        scheduled.showDateTime(),
                        scheduled.doorsTime(),
                        scheduled.ticketPrice()));
                ticketsLeftByConcertId.put(scheduled.concertId(), scheduled.capacity());
            }
            case ConcertRescheduled rescheduled -> concerts.computeIfPresent(
                rescheduled.concertId(),
                (_, concert) -> new Concert(concert.concertId(),
                    concert.artist(),
                    rescheduled.newShowDateTime(),
                    rescheduled.newDoorsTime(),
                    concert.ticketPrice()));
            case TicketsSold sold -> ticketsLeftByConcertId.computeIfPresent(
                sold.concertId(),
                (_, ticketsLeft) -> ticketsLeft - sold.quantity());
            default -> {} // Read, but not applied: this read model doesn't care about it.
        }
    }
}

```

Which Concerts Have Tickets Left?



One Concept, Many Uses

The job	ORM	Event Sourcing
State to decide with	load the aggregate	a projection
A UI screen	DTO + mapper	a projection
An API payload	another DTO + serializer	a projection
A table for another system	denormalised table + sync job	a projection
A report	materialised view + refresh schedule	a projection

Five jobs. Five mechanisms — or **one, used five times.**



Event Store

Every application decision outcome stored in strict serial order.

Append-Only



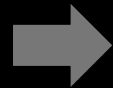
Events are never changed nor deleted,
nothing can be lost

Ordered



Events always "replay" to the same state

Queryable



Load **only** the events a decision needs
(we'll use this later)

Remembering history doesn't need to be bolted on, it's already there.



Just a Log: a Table in a Database

```
CREATE TABLE IF NOT EXISTS events (  
  -- Global, gap-tolerant position that supports strict ordering,  
  -- projection checkpoints, and conditional-append check  
  event_sequence BIGINT GENERATED ALWAYS AS IDENTITY,  
  -- Entity that "owns" the resource modified by the event  
  owner_id      UUID          NOT NULL,  
  PRIMARY KEY (owner_id, event_sequence),  
  -- Unique ID supports idempotence: retried appends are detected and skipped  
  event_id      UUID          NOT NULL,  
  -- Name of the event, not a fully-qualified class name  
  event_type    TEXT          NOT NULL,  
  -- DCB tags, e.g. '{con:989-ZZX, cust:123-ABC}' supporting querying  
  tags          TEXT[]        NOT NULL DEFAULT '{}',  
  -- Event payload  
  payload       JSONB         NOT NULL,  
  -- Per-event schema version for migration support  
  schema_version INTEGER      NOT NULL,  
  created_at    TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP  
);
```



Event Sourcing is Not...



**Event-Driven
Architecture**

**Integrate across
Context Boundaries**



Event Streaming

High-volume/rate of Events



**Command-Query
Responsibility Separation**

**Required by Event-
Sourcing, but not vice-versa**



DDD Tactical Patterns (quick refresher)

Entity

Identity, history, attributes

Event-Sourcing:
retains history entities have
but rarely saved.

Value Object

Immutable, compared by
attributes

Event-Sourcing:
Love 'em! Used in events and
in (projected) state.

Aggregate

Consistency (transactional)
boundary

Event-Sourcing:
boundary is a query instead
of a rigid object.

Repository

Store & retrieve aggregates

Event-Sourcing:
Replaced by Event Store and
read models.



Where Do Decisions Live?

State + [Context] + Command Request -> Decider -> Event

Definition: DECIDER

The heart of every Command.

**Given a *Decision Context*,
decides whether to generate an
Event or reject the Command.**



Deciders Are Pure

Pure Domain, No I/O (No fetching, UUID, Clock)

Event Store -> folded to STATE

Event Store and/or I/O -> additional CONTEXT

triggering information -> command REQUEST



**Decision Context
for the Decider**

Events = State + Command + Decider

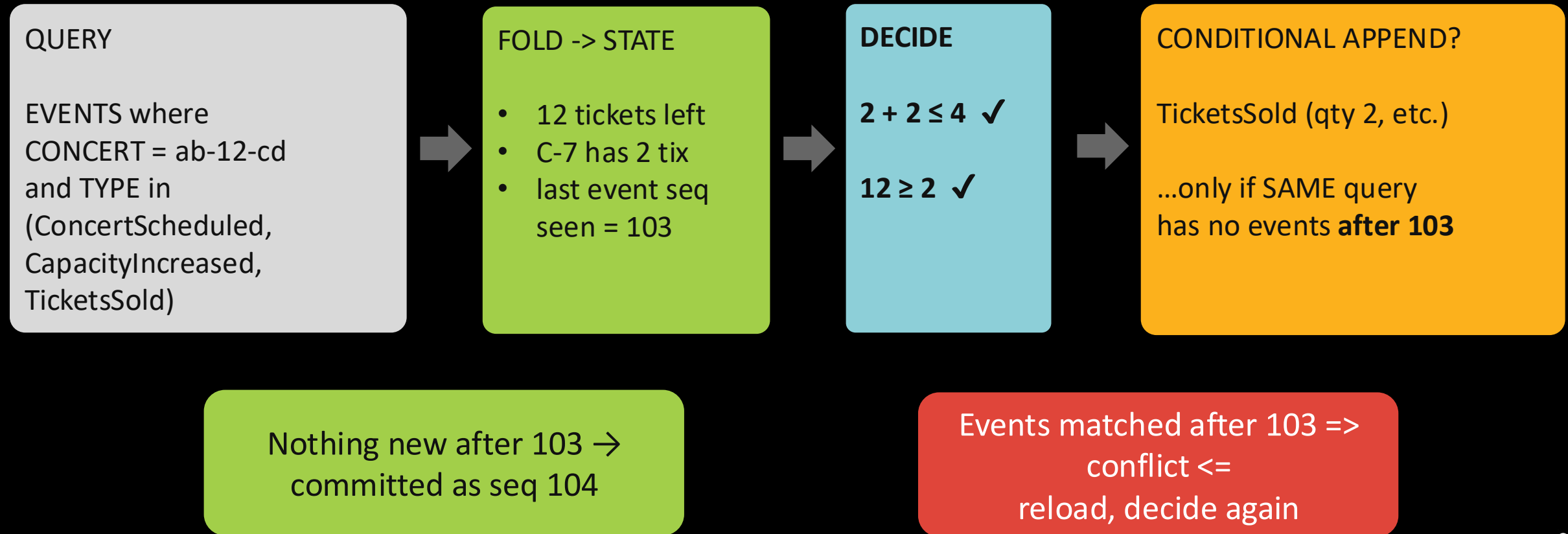
```
private String artist;  
private int ticketPrice;  
private LocalDateTime showDateTime;  
private LocalTime doorsTime;  
private int capacity;  
private int maxTicketsPerPurchase;  
private int availableTicketCount;  
private boolean ticketsOnSale = true;
```

```
public void rescheduleTo(LocalDateTime newShowDateTime, LocalTime newDoorsTime) {  
    if (newShowDateTime.isAfter(showDateTime) && capacity - availableTicketCount == 0) {  
        ConcertRescheduled concertRescheduled =  
            new ConcertRescheduled(getId(), null, newShowDateTime, newDoorsTime);  
        enqueue(concertRescheduled);  
    }  
}
```

Consistency Without Aggregates

Rule: a customer may buy at most 4 tickets per concert.

Customer C-7 wants 2 more for concert ab-12-cd.



What The Decision Query Returns

Two filters at work: tag picks the Concert, type list picks what we care about

seq	event_type	concert_id	customer_id	payload	
23	ConcertScheduled	ab-12-cd	—	capacity 100 · maxPerCustomer 4 · price 135.00	✓
41	~~TicketPriceChanged~~	ab-12-cd	—	~~ticketPrice 35.00~~	not in the type list
47	TicketsSold	ab-12-cd	C-3	quantity 4	✓
58	TicketsSold	ab-12-cd	C-7	quantity 2	✓
66	~~ArtistChanged~~	ab-12-cd	—	~~artist → "Soul Loop II"~~	not in the type list
73	CapacityIncreased	ab-12-cd	—	capacity 120	✓
74-95	TicketsSold × 18	ab-12-cd	various	93 tickets — none of them C-7's	✓
96	TicketsSold	ab-12-cd	C-12	quantity 6	✓
103	TicketsSold	ab-12-cd	C-41	quantity 3	✓
104	~~TicketsSold~~	~~xy-98-zz~~	C-7	~~quantity 2~~	different concert

Event 104 is NEWER than the 103 we read — yet does not conflict: does not match our query.

Not “has anything happened?” but “has anything happened that would have changed THIS decision?”



One List, Two Numbers

Same events, two accumulators, one pass — and only one row moves both.

seq	event	tickets available	C-7 holds
23	ConcertScheduled — capacity 100, max per customer 4	100	0
47	TicketsSold C-3 × 4	96	0
58	TicketsSold C-7 × 2 ← only row that affects both	94	2
73	CapacityIncreased — capacity 120	114	2
74–95	TicketsSold × 18 — 93 tickets, none to C-7	21	2
96	TicketsSold C-12 × 6	15	2
103	TicketsSold C-41 × 3	12	2

12 available · C-7 holds 2 · limit 4 · sales open · last matching seq 103

→

$12 \geq 2 \checkmark \quad 2 + 2 \leq 4 \checkmark$



The Fold, in Code

```
record State(int capacity, int sold, int customerHeld,
             int maxPerCustomer, boolean salesOpen) {
    int available() { return capacity - sold; } // derived, not stored
}

State fold(List<Event> events, CustomerId customer) {
    var state = State.EMPTY;
    for (Event e : events) {
        state = switch (e) {
            case ConcertScheduled c    -> s.capacity(c.capacity())
                                     .maxPerCustomer(c.maxTicketsPerCustomer());
            case CapacityIncreased c    -> s.capacity(c.capacity());
            case TicketSalesStopped _  -> s.salesClosed();
            case TicketsSold t         -> t.customerId().equals(customer)
                                     ? s.addSold(t.quantity()).addHeld(t.quantity())
                                     : s.addSold(t.quantity());
        };
    }
    return state;
}
```

- customer is an argument to the fold — not part of the query.
- Only TicketsSold branch uses the customer. Everything else is concert-wide.
- available() is derived. Capacity and Sold are facts, availability is math.



Still Need Aggregates?

Aggregates + Entities can still be useful

Soften the Decision Boundaries

Benefits of Event-Sourcing

Ask Questions You Didn't Know You Had

How many Customers buy maximum # of Tickets?

How often are we Rescheduling Concerts?

Do we ever reduce Maximum Capacity?

Data Shape Matches Questions Precisely

Relationships between customers = Graph projection

Music recommendations = Vector projection

API for external usage = JSON in-memory projection

Consistent Test Structure (Events→Cmd→Event)

- All **Command-oriented** tests are:
 - GIVEN: Event(s)
 - WHEN: execute Command
 - THEN: expect Event

(ConcertTest.rescheduleConcertGeneratesConcertRescheduled)



@Test

```
void rescheduleConcertGeneratesConcertRescheduled() {  
    LocalDateTime originalShowDateTime = LocalDateTime.of(2025, 11, 11, 20, 0);  
    LocalTime originalDoorsTime = LocalTime.of(19, 0);  
    ConcertScheduled concertScheduled =  
        ConcertFactory.Events.scheduledConcert(originalShowDateTime, originalDoorsTime);  
    Concert concert = Concert.reconstitute(List.of(concertScheduled));  
  
    LocalDateTime newShowDateTime = originalShowDateTime.plusDays(1).minusHours(1);  
    LocalTime newDoorsTime = originalDoorsTime.minusHours(1);  
    concert.rescheduleTo(newShowDateTime, newDoorsTime);  
  
    assertThat(concert.uncommittedEvents())  
        .containsExactly(new ConcertRescheduled(  
            concert.getId(), null,  
            newShowDateTime,  
            newDoorsTime));  
}
```



@Test

```
void executeJoinGameReturnsSuccessfulPlayerJoinedEvent() {  
    EventStore eventStore = new InMemoryEventStore();  
    GameHandle gameHandle = new GameHandle("game to be joined");  
    eventStore.append(EventFactory.gameCreatedWithHandle(gameHandle));  
    MemberRegistered memberRegistered = EventFactory.memberRegistered();  
    eventStore.append(memberRegistered);  
    JoinGameCommand joinGameCommand = new JoinGameCommand(eventStore);  
  
    Result<Event, String> result = joinGameCommand.execute(  
        memberRegistered.memberId(),  
        gameHandle);  
  
    EventResultAssert.assertThat(result)  
        .succeeded()  
        .isOfType(PlayerJoined.class)  
        .containsTagValues(gameHandle, memberRegistered.memberId());  
}
```



Consistent Test Structure (Events→Projection)

- All **State-oriented** (projection) tests are:
 - GIVEN: Event(s)
 - THEN: expected State

(ConcertSalesProjectorTest.multipleConcerts_MultipleTicketsSoldPerConcert_ProjectsRowPerConcert)



Easier Troubleshooting

Discover a Bug? See the trail of decision outcomes.

Customized Projections

Disposable: easy to create, change, and recreate

'Immutable' Events*

Never lose information: there is no DELETE

* Events might change as part of Event Migration

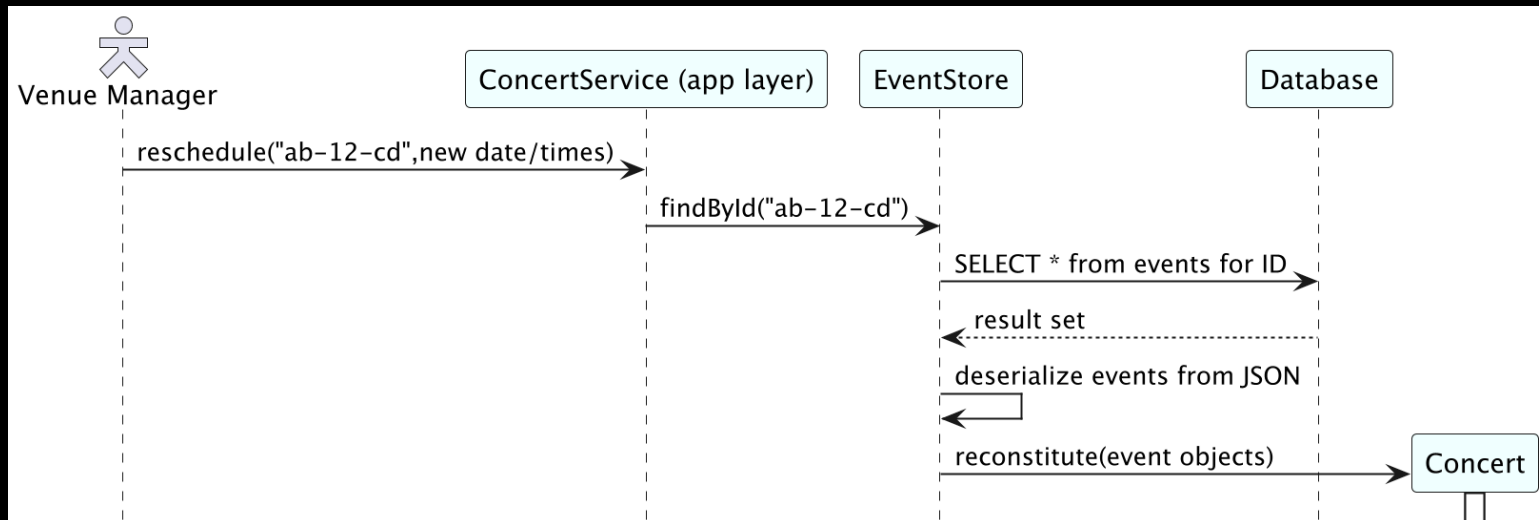
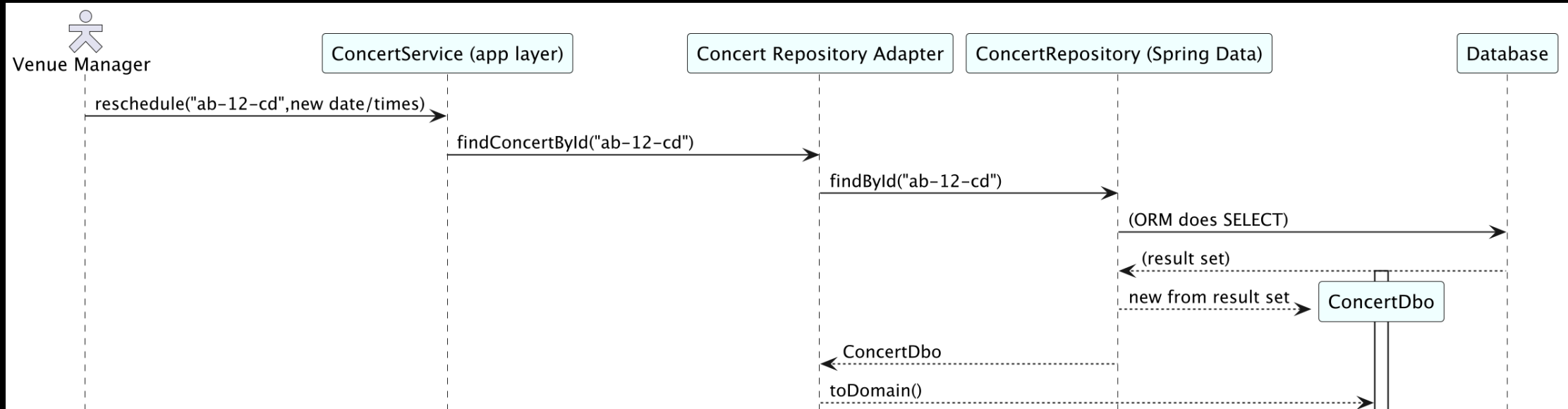
Fewer Concepts

Commands, Events, Projections

Less complexity than ORMs

No awkward mapping, lazy loading, or dirty-checking

Simpler Steps



Replayability

Time Travel: see state of system in the past

Let's talk about Performance

Similar, if not BETTER than ORM

Startup New Projections Creation

1 Million (1,000,100) Events:

11:41:16.848 ProjectionCoordinator : Catching up for AvailableConcertsProjector
11:41:16.870 JdbcEventStore : Fetching [ConcertRescheduled, TicketSalesStopped, ConcertScheduled, TicketsSold], after event sequence: C
11:41:39.447 JdbcEventStore : Fetched 1,000,100 events, mapping to Domain events
11:41:39.451 ProjectionCoordinator : Fetched events, now updating projection...
11:41:43.562 ProjectionCoordinator : Subscribing to event store for all events
11:41:43.581 ProjectionCoordinator : Catching up for ScheduledConcertsProjector
11:41:43.593 JdbcEventStore : Fetching [ConcertRescheduled, TicketSalesStopped, ConcertScheduled, TicketsSold], after event sequence: C
11:42:06.159 JdbcEventStore : Fetched 1,000,100 events, mapping to Domain events
11:42:06.159 ProjectionCoordinator : Fetched events, now updating projection...
11:42:10.406 NewProjectionCoordinator : Catching up on events for event types: [class CustomerRegistered]
11:42:10.406 JdbcEventStore : Fetching [CustomerRegistered], after event sequence: Checkpoint(0/INITIAL)
11:42:10.641 JdbcEventStore : Fetched 10,000 events, mapping to Domain events
11:42:10.641 NewProjectionCoordinator : Fetched events, now handling them...
11:42:10.700 ProjectionCoordinator : Catching up for AllConcertsProjector
11:42:10.719 JdbcEventStore : Fetching [ConcertRescheduled, TicketSalesStopped, ConcertScheduled, TicketsSold], after event sequence: C
11:42:33.915 JdbcEventStore : Fetched 1,000,100 events, mapping to Domain events
11:42:33.915 ProjectionCoordinator : Fetched events, now updating projection...
11:42:38.036 ProcessorConfiguration : Concert Started Processor: catching up on all events
11:42:38.036 JdbcEventStore : Fetching [TicketSalesStopped, ConcertScheduled, ConcertRescheduled], after event sequence: Checkpoint(0/
11:42:38.042 JdbcEventStore : Fetched 100 events, mapping to Domain events



ORM Pros/Cons

- It's familiar
- Solid tooling
- DB enforces constraints
- Writes can be read immediately
- History thrown away
- Audit tables lossy and guesswork
- Mapping, Caching, N+1, DTOs, Sagas...



Event Sourcing Pros/Cons

- Full history kept
- No guessing about the future
- One mechanism, Projections, handles many needs
- No mapping layer
- Mindset shift!
- Read Models can lag (eventual consistency)
- Can't do ad-hoc queries



When NOT to use Event Sourcing?

- **Real CRUD: users edit tables, no domain logic**
- **Must use shared, legacy tables**
- **Nobody will ever care about history**
- **Existing system: migration too costly**
 - **Use for new Bounded Contexts**



Your Next Steps...

- **Write an event-sourced app From Scratch**
 - It's not a lot of code
- **Libraries and frameworks for Production**
 - Java: Axoniq and FactCast
 - .NET: Marten
 - TypeScript: Emmett
 - PHP: EventSauce, Patchlevel



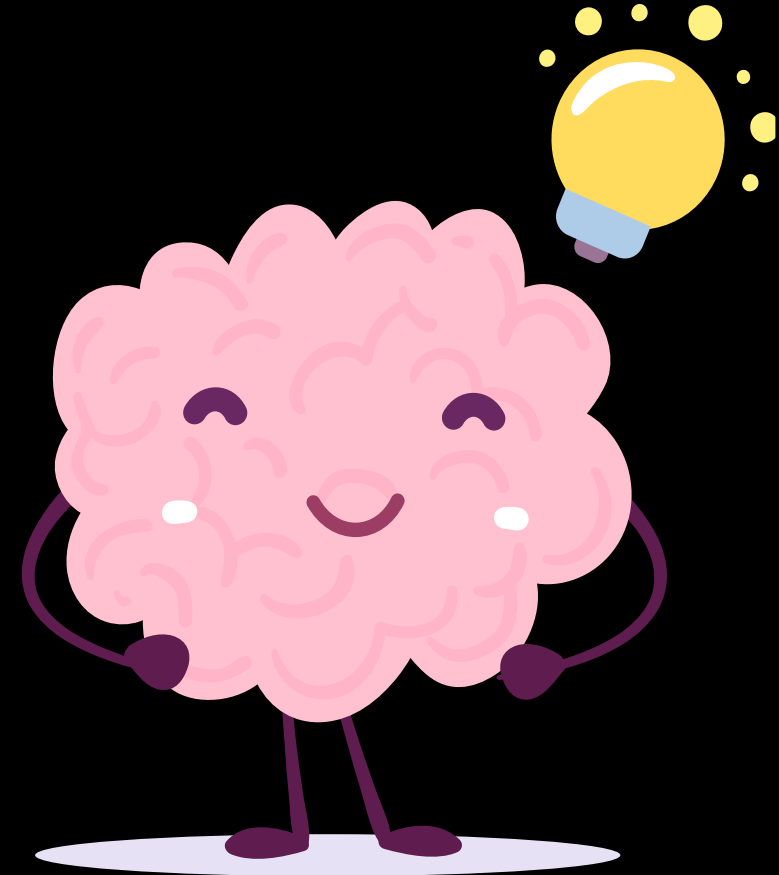
Ted M. Young

Java Trainer, Coach, Live Coder,
and Creator of JitterTed's TDD Game

Get in touch: ted@tedmyoung.com

About me: <https://ted.dev/about>

Final Questions?



Ted M. Young

Java Trainer, Coach, Live Coder,
and Creator of JitterTed's TDD Game

Get in touch: ted@tedmyoung.com

About me: <https://ted.dev/about>

Thank You...

Source Code? Slides?

<https://ted.dev/talks/>

Please provide feedback here:



Other Topics...

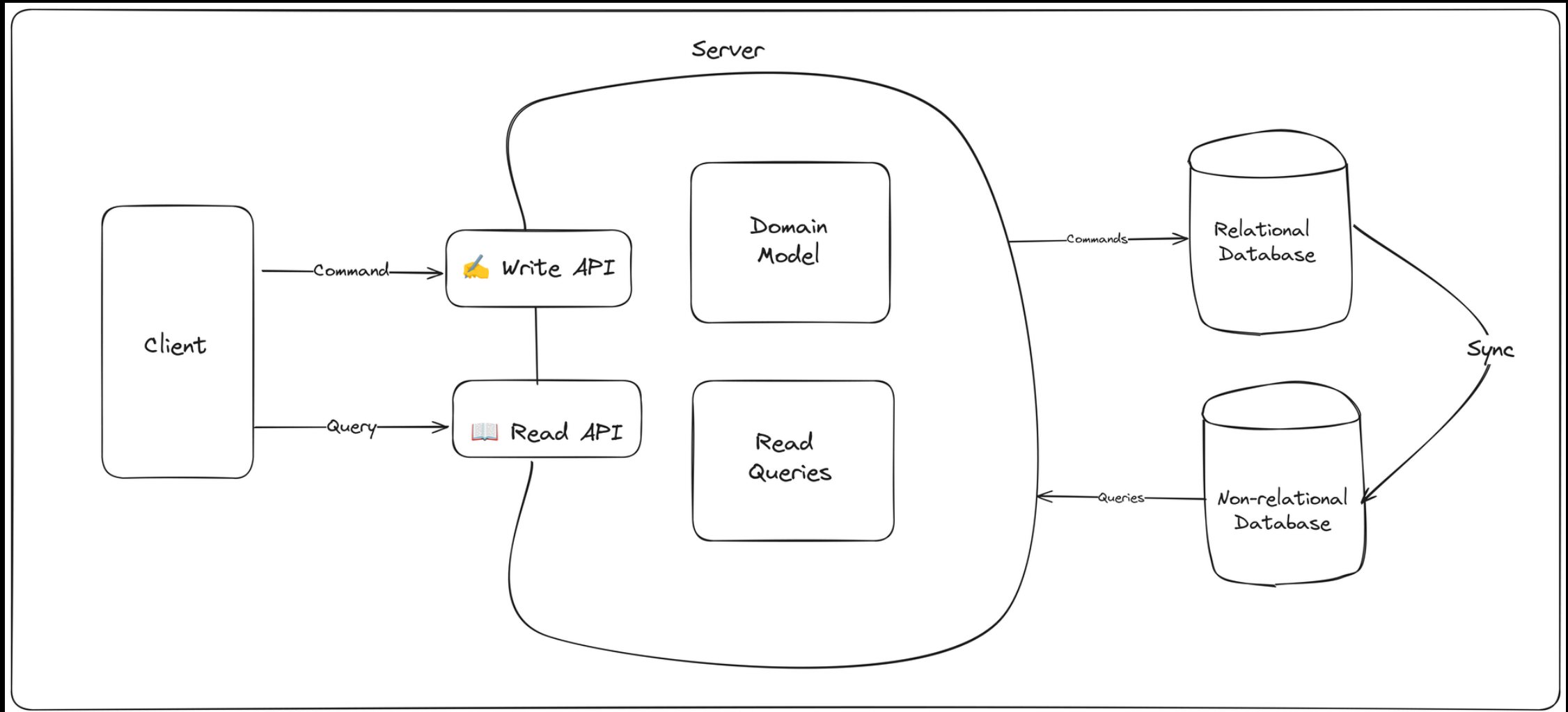
Versioning Event Logs (Streams)

(see Greg Young's Versioning in an Event-Sourced System)

- **Copy-Replace**
- **Split-Stream**
- **Join-Stream**
- **Copy-Transform**



CQRS – Separated Data Models



```
graph TD; Command[Command] --> Event[Event]; Event --> State[State];
```

The diagram illustrates a three-step process flow. It begins with a blue rectangular box labeled "Command". An arrow points from the bottom of the "Command" box to the top of an orange rectangular box labeled "Event". Another arrow points from the bottom of the "Event" box to the top of a green rectangular box labeled "State".

